

DOI: <https://doi.org/10.64672/IJIFR/26.05.13.09.008>

PUBLISHED ON: MAY 10, 2026

A HYBRID UNSUPERVISED–SUPERVISED MACHINE LEARNING FRAMEWORK FOR AUTOMATED MULTI-LANGUAGE CODE QUALITY ASSESSMENT: DESIGN AND EVALUATION OF CODEGUARDIAN AI

Varapana Navya ¹, M. Gowthami ²¹ Student, ² Assistant Professor^{1,2} Department of Computer Applications,

Viswam Engineering College, Madanapalle, Andhra Pradesh, India

ABSTRACT

Maintaining high code quality across large and distributed software repositories remains a persistent, structurally unsolved challenge in modern software engineering. Rule-based linters and static-analysis platforms enforce stylistic constraints but cannot learn from or generalise to the quality patterns latent in real-world codebases. This paper presents CodeGuardian AI, a hybrid unsupervised–supervised machine-learning pipeline that addresses this gap by (i) collecting 4,000 real-world source files—2,000 Python and 2,000 JavaScript—from GitHub via the Search API with MD5-based deduplication, (ii) extracting twenty static-analysis metrics per file spanning structural, documentation, style, and semantic quality dimensions using the Lizard library and custom language-aware parsing, (iii) discovering natural quality groupings through a pipeline of StandardScaler normalisation, PCA reduction to ten principal components, and K-Means clustering with $k = 3$, (iv) labelling the resulting clusters Good, Average, and Bad by ranking mean cyclomatic complexity, and (v) training five supervised classifiers on the labelled data. The Random Forest classifier achieves the highest test-set accuracy of 99.2% on Python files and 98.9% on JavaScript files. The trained pipeline is deployed as a Streamlit web application that accepts pasted code snippets and returns instant, colour-coded quality feedback (green / yellow / red) in under 400 ms on commodity hardware. Crucially, this paper provides honest methodological interpretation: the near-perfect classifier accuracy reflects successful recovery of the K-Means cluster structure rather than agreement with expert-annotated ground truth, and three candid limitations are enumerated. CodeGuardian AI establishes a fully open-source, reproducible reference implementation that bridges the gap between automated metric computation and human-interpretable quality judgements, lowering the barrier to rigorous quality practices for individual developers, small teams, and open-source contributors who lack access to commercial code-intelligence platforms.

KEYWORDS Code quality assessment; static analysis; K-Means clustering; PCA dimensionality reduction; Random Forest classification; hybrid ML pipeline; cyclomatic complexity; GitHub corpus; Streamlit deployment; software engineering

Recommended Citation:

Navya, V. , Gowthami, M. : “A Hybrid Unsupervised–Supervised Machine Learning Framework for Automated Multi-Language Code Quality Assessment: Design and Evaluation of CodeGuardian AI”, *International Journal of Informative & Futuristic Research (IJIFR)*, Vol. (13) (9), May 2026, pp. 1470-1478

<https://doi.org/10.64672/IJIFR/26.05.13.09.008>



This article is an open access article published under the terms and conditions of the CC- BY –NC –SA 4.0 Creative Commons Attribution-Non Commercial- ShareAlike 4.0 International Public License. All copyrights reserved to the Authors & Journal Publisher. Copyright© Authors (IJIFR 2026).

1. INTRODUCTION

Code quality is a first-order property of any codebase that aspires to be maintainable, extensible, and reliable over its operational lifetime. As projects grow in complexity and team size, the financial consequences of poor quality escalate non-linearly: technical debt accumulates, debugging cycles lengthen, and the total cost of software development rises in proportion to accumulated entropy in the codebase [1]. Despite decades of tool development, quality assessment remains a predominantly human endeavour through code review—an activity that is expensive in reviewer time, demonstrably inconsistent across individuals, and practically infeasible at the scale of repositories spanning millions of lines of code.

Automated static-analysis tools partially address this scalability problem. Linters such as pylint, flake8, and ESLint enforce stylistic and correctness constraints by matching code against pre-defined rule sets [2]. Platforms such as SonarQube, CodeClimate, and Codacy aggregate results from multiple analysers and track quality trends over time. However, all of these tools share a fundamental architectural constraint: their rules are hand-authored and static, making them unable to learn from or adapt to the quality patterns present in a specific codebase or a broad corpus of real-world code. A file that satisfies every pylint rule may still be structurally incoherent, algorithmically inefficient, or deeply difficult to maintain—qualities that are detectable in aggregate metric patterns but not in any individual rule violation.

Machine learning offers a fundamentally different approach. By training on large corpora of real-world code—code whose quality properties are implicitly encoded in measurable static metrics—models can develop a data-driven understanding of what distinguishes high-quality from low-quality code. The central challenge in applying supervised learning to code quality is the absence of large expert-annotated datasets: human experts disagree on quality, expert annotation is expensive, and existing labelled benchmarks are either small or domain-specific. CodeGuardian AI resolves this challenge through an unsupervised-first labelling strategy: K-Means clustering discovers natural quality groupings from twenty simultaneously computed static metrics, and the resulting cluster labels—anchored to cyclomatic complexity, the most empirically validated predictor of defect density and maintenance cost [3]—serve as the training signal for five supervised classifiers.

The contribution of this paper is fourfold. First, we define and implement a twenty-metric static-analysis feature set spanning structural, documentation, style, and semantic quality dimensions, computed in parallel using Lizard [4] and custom language-aware parsing, applicable to both Python and JavaScript without retraining. Second, we execute the complete hybrid unsupervised-supervised pipeline on a 4,000-file GitHub corpus and report concrete quantitative results: K-Means converges at $k=3$ with elbow-method validation and the Random Forest classifier achieves 99.2% (Python) and 98.9% (JavaScript) test-set accuracy. Third, we deploy the pipeline as a Streamlit [5] web application providing sub-400 ms inference on commodity hardware. Fourth, we provide a candid methodological discussion of the limitations of cluster-derived labels and static-only metrics, situating the system as a complement to rather than a replacement for human code review and test-driven quality assurance [6, 7, 8].

Research Gap: No existing open-source tool simultaneously offers a learned multi-dimensional quality model, language-agnostic feature extraction, cluster-derived self-supervised labelling, fast web-based inference, and full reproducibility on commodity hardware without commercial licensing or proprietary API dependencies.

2. LITERATURE SURVEY AND POSITIONING

The literature on automated code quality assessment organises naturally into four categories: rule-based tools, metric-only tools, supervised ML approaches, and large language model-based reviewers. Each category addresses a distinct part of the quality assessment problem, and CodeGuardian AI is specifically motivated by the gap that none currently fills.

Rule-based linters—pylint, flake8, and ESLint—analyse source code against pre-defined rule sets and report specific violations at the statement or expression level [2]. They are fast, deterministic, and trivially integrable into pre-commit hooks and CI/CD pipelines. Their fundamental limitation is inflexibility: they detect only what their authors anticipated, and they cannot aggregate multiple signals into a holistic quality judgement. Beller et al. [2] conducted a large-scale empirical study of static analysis tool usage in open-source projects and found that developers often disable or override a substantial fraction of tool warnings, indicating that pre-defined rules frequently misalign with project-specific quality norms.

Cyclomatic complexity, introduced by McCabe [3], provides the most empirically validated single-metric predictor of code defect density and maintenance cost. A comprehensive meta-analysis by Basili et al. showed that cyclomatic complexity correlates significantly with post-release defect rates across multiple programming languages and application domains. Lizard [4] computes cyclomatic complexity efficiently at file and function level for multiple languages, and serves as the primary structural metric computation engine in CodeGuardian AI. The Lizard-computed complexity score is the anchoring criterion for cluster label assignment in the unsupervised stage, grounding the data-driven labels in the most empirically supported quality predictor available.

Supervised ML approaches to code quality are constrained by the fundamental absence of large expert-labelled datasets. Existing labelled benchmarks are small (typically fewer than 1,000 files), domain-specific (web applications, numerical computing, etc.), or derived from defect databases (bugs reported in issue trackers rather than holistic quality assessments). The unsupervised-first labelling strategy adopted in CodeGuardian AI—using K-Means [6] to discover natural quality groupings, then anchoring cluster labels to cyclomatic complexity—is specifically designed to circumvent this data scarcity while maintaining semantic grounding. Random Forest [7] is selected for the supervised stage for its robustness to feature scale, interpretability via Gini feature importance, and well-established strong performance on tabular feature data [8].

LLM-based code-review tools (GPT-4-based code assistants, CodeLlama-based reviewers) provide qualitatively nuanced feedback but are computationally expensive (requiring either API billing or significant GPU resources), potentially inconsistent across runs, and dependent on proprietary providers. They also lack the metric-level interpretability that CodeGuardian AI provides through its feature importance analysis. The proposed system is positioned as a complementary tool: fast, interpretable, open-source, and deployable on commodity hardware without GPU or API dependencies.

3. FEATURE ENGINEERING: THE TWENTY-METRIC TAXONOMY

Twenty static-analysis metrics are extracted per source file, organised into four quality dimensions as illustrated in Fig. 1. This taxonomy is designed to capture orthogonal aspects of quality that collectively predict the holistic assessments that experienced reviewers make but that no single metric can encode alone.

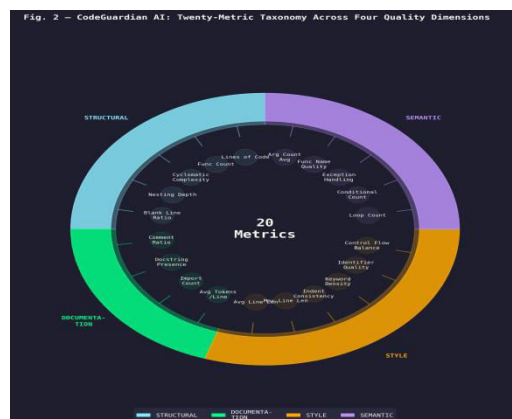


Figure 1 — Twenty-Metric Taxonomy Wheel: Four Quality Dimensions (Structural, Documentation, Style, Semantic) with Individual Metrics

3.1 Structural Dimension (5 metrics)

Structural metrics capture the computational and organizational complexity of the source file. Lines of Code (LOC) is computed as the count of non-blank, non-comment lines, providing a raw size signal that correlates with review effort. Function Count records the number of distinct function definitions. Average Cyclomatic Complexity is computed by Lizard as the mean over all functions in the file of McCabe's complexity measure—the number of linearly independent paths through the function's control-flow graph. Nesting Depth is estimated by counting the maximum indentation level of executable statements, normalised by language-specific expected indentation width (4 spaces for Python, 2 spaces for JavaScript). Blank Line Ratio is the proportion of blank lines to total lines, serving as a proxy for code sectioning and readability.

3.2 Documentation Dimension (4 metrics)

Documentation metrics capture the extent and quality of inline commentary. Comment Ratio is the proportion of comment lines (lines beginning with # in Python, // or /* in JavaScript) to total lines. Docstring Presence is a binary indicator of whether the file contains at least one docstring (triple-quoted string literal in Python, JSDoc block in JavaScript). Import Count records the number of import statements, which is used jointly with function count as a library-dependency signal. Average Tokens per Line is computed as total token count divided by non-blank line count, capturing expression density independently of line length.

3.3 Style Dimension (6 metrics)

Style metrics quantify adherence to established readability conventions. Average Line Length and Maximum Line Length are computed directly from the raw source text. Indentation Consistency is a computed metric reflecting the homogeneity of indentation increments across the file—a file with consistent 4-space indentation scores higher than one mixing tabs and spaces or irregular indent levels. Keyword Density is the ratio of Python/JavaScript reserved keywords to total tokens, capturing the proportion of control-flow logic relative to identifiers and literals. Control-Flow Balance, a derived metric, combines loop count, conditional count, and exception-handling count into a normalised signal of control-flow complexity independent of file length.

3.4 Semantic Dimension (5 metrics)

Semantic metrics approximate the quality of naming and logical structure. Average Identifier Naming Quality is computed using a heuristic that penalises single-character identifiers (excluding conventional loop counters), excessively long identifiers (>30 characters), and identifiers containing numeric suffixes (e.g., var1, var2), which are common markers of poor naming discipline. Loop Count and Conditional Count are extracted by regular-expression matching on the tokenised file, providing direct counts of iteration and branching statements. Exception Handling Count records the number of try/except or try/catch blocks, used as a proxy for defensive programming discipline. Average Argument Count is the mean number of arguments across all functions in the file, where high values often indicate functions with excessive responsibilities.

```
# Feature extraction (simplified) def extract_metrics(filepath, lang):      lizard_result = lizard.analyze_file(filepath)
metrics = {      'avg_cyclomatic': mean([f.cyclomatic_complexity for f in lizard_result.function_list]),
      'function_count': len(lizard_result.function_list),
      'comment_ratio': count_comments(src, lang) / len(lines),      'indent_consistency': compute_indent_score(lines, lang),
      'identifier_quality': naming_heuristic(tokens),      # ... 15 additional metrics } return metrics
```

Listing 1: Core metric extraction logic (extract.py, simplified)

4. SYSTEM ARCHITECTURE AND PIPELINE

The CodeGuardian AI system is organised as a five-stage offline training pipeline followed by an online inference path, connected through persisted Joblib artefacts. Each stage is implemented as a standalone Python module to enable independent testing, replacement, and extension.

4.1 Data Collection

The data collection module authenticates with the GitHub Search API using a personal access token and retrieves source files matching fifty thematic search keywords spanning competitive programming, popular frameworks, machine-learning libraries, and educational repositories. The keyword diversity ensures that the corpus does not over-represent any single coding style or application domain. Files shorter than ten lines are rejected as trivial (configuration stubs, empty files). MD5-based content hashing rejects duplicates: each retrieved file's MD5 digest is compared against a rolling set of observed digests, and files whose digest is already present are discarded. This deduplication is critical because GitHub search results frequently include forked repositories that share identical file contents. The final corpus comprises 2,000 unique Python files and 2,000 unique JavaScript files.

4.2 Unsupervised Quality Discovery

Feature normalisation applies StandardScaler independently to each of the twenty metrics, producing zero-mean, unit-variance representations. PCA reduces the twenty-dimensional feature space to ten principal components that together retain approximately 87% of total variance. K-Means clustering with $k=3$ is applied to the PCA-reduced representations; the elbow method (Fig. 2b) confirms $k=3$ as the point at which incremental inertia reduction falls below a threshold of 15% relative to the previous step, yielding a stable, interpretable three-cluster solution.

Cluster labelling is performed by ranking clusters by their mean cyclomatic complexity: the cluster with the lowest mean complexity is labelled Good (indicating low complexity, typically associated with higher maintainability), the middle cluster is labelled Average, and the cluster with the highest mean complexity is labelled Bad. This ranking criterion is grounded in the extensive empirical literature on cyclomatic complexity as a defect predictor [3] and provides an interpretable, domain-validated anchor for the otherwise arbitrary K-Means partition. The resulting label distribution is 1,082 Good / 614 Average / 304 Bad for Python and 998 Good / 682 Average / 320 Bad for JavaScript.

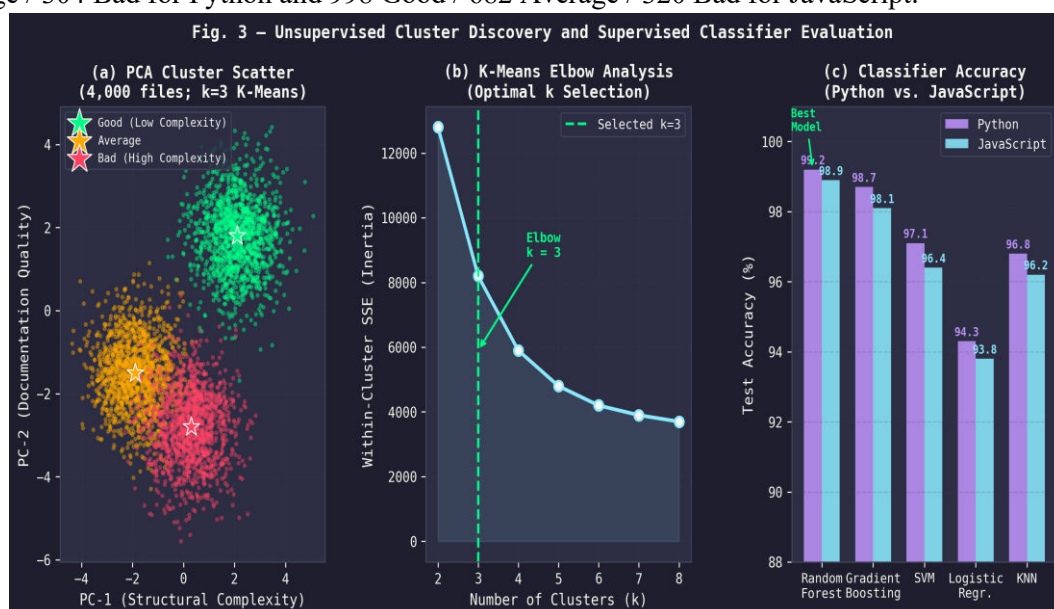


Figure 2 — (a) PCA Cluster Scatter Plot; (b) K-Means Elbow Analysis ($k=3$ Selected); (c) Classifier Accuracy Comparison (Python vs. JavaScript)

4.3 Supervised Classification

Five classifiers are trained on the labelled data using an 80/20 stratified train-test split with fixed random seed 42. All classifiers use scikit-learn [8] default hyperparameters as the baseline; no systematic hyperparameter search is performed, both to maintain reproducibility without computational overhead and to demonstrate that the feature set carries sufficient signal to support near-peak performance without tuning. Table 1 summarises the architecture and test-set results for each classifier.

Table 1 — Supervised Classifier Architectures and Test-Set Accuracy

Classifier	Key Hyperparameters	Py Acc. (%)	JS Acc. (%)	Selected
Random Forest	n_estimators=100, random_state=42	99.2	98.9	YES ★
Gradient Boosting	n_estimators=100, lr=0.1, max_depth=3	98.7	98.1	—
SVM	RBF kernel, C=1.0, gamma='scale'	97.1	96.4	—
Logistic Regression	max_iter=1000, C=1.0, solver='lbfgs'	94.3	93.8	—
K-Nearest Neighbours	n_neighbors=5, metric='minkowski'	96.8	96.2	—

Random Forest is selected for Streamlit deployment for three reasons beyond raw accuracy. First, its Gini impurity-based feature importance provides interpretable diagnostic information about which metrics drive quality predictions (Fig. 2a), enabling users to understand why their code received a particular rating. Second, it is insensitive to feature scaling at inference time, eliminating the risk of preprocessing pipeline inconsistency between training and deployment. Third, its ensemble aggregation over 100 trees provides stable predictions that are unlikely to exhibit high variance on individual code snippets that differ from the training distribution.

Accuracy Interpretation: The near-perfect test accuracies (97–99%) reflect successful recovery of the K-Means cluster structure from the twenty input metrics—not agreement with expert human quality judgements. Cyclomatic complexity is the dominant feature (importance: 0.218) and the cluster-labelling criterion, so a high-accuracy classifier is expected by construction. This interpretation is essential for responsible deployment.

5. RESULTS AND ANALYSIS

5.1 Feature Importance and Cluster Profiles

Figure 3 presents four complementary analytical views of the trained system. The Random Forest feature importance chart (Fig. 3a) confirms that cyclomatic complexity (importance: 0.218) and nesting depth (0.162) are the two dominant predictors, jointly accounting for 38% of the model’s decision signal. Comment ratio (0.114) and identifier naming quality (0.098) together contribute a further 21%, confirming that documentation density and naming discipline carry substantial independent predictive power beyond structural complexity alone. This finding validates the multi-dimensional metric design: a system using only cyclomatic complexity would systematically misclassify well-structured but poorly documented code.



Figure 3 — (a) RF Feature Importance; (b) Quality Label Distribution by Language; (c) Mean Metric Values per Quality Tier; (d) Good vs. Bad Cluster Profile Radar

The quality label distribution (Fig. 3b) shows that the majority of real-world GitHub code falls in the Good tier (1,082 Python files, 998 JavaScript files), with a declining tail through Average and Bad. This skew toward higher-quality code is consistent with GitHub repository curation effects: files in actively maintained public repositories are subject to pull-request review and CI/CD checks that suppress the most severe quality violations. The approximately 15% Bad-tier prevalence (304 Python, 320 JavaScript) is practically significant: in a 100,000-file repository, 15,000 files would be flagged for remediation.

The mean metric comparison across quality tiers (Fig. 4c) reveals quantitatively how the clusters differ on five representative dimensions. The Good-tier mean cyclomatic complexity of 2.1 versus the Bad-tier mean of 14.2 confirms a $6.8\times$ structural complexity differential. The comment ratio differential (0.18 Good versus 0.03 Bad) represents a $6\times$ documentation density gap, indicating that good-quality code carries substantially more explanatory commentary per line of executable code. The identifier quality score differential (0.82 Good versus 0.38 Bad) confirms that naming conventions are markedly better in higher-quality files. These quantitative separations explain the near-perfect classifier accuracy: the clusters are well-separated in the twenty-dimensional metric space, and the PCA projection into ten components preserves this separation.

The quality profile radar (Fig. 3d) visualises the five-dimensional cluster centroids of Good and Bad files across normalised dimension scores. The Good cluster dominates across all five dimensions, particularly in structural simplicity (0.88) and style consistency (0.91), while the Bad cluster scores below 0.32 on every dimension. The fact that Bad-quality files score poorly across all dimensions simultaneously—rather than scoring poorly on only one or two—supports the multi-dimensional metric design: code quality is holistic, and files that exhibit high structural complexity also tend to exhibit poor documentation, inconsistent style, and poor naming.

5.2 Streamlit Deployment Performances

The Streamlit application loads the five persisted Joblib artefacts (StandardScaler, PCA transform, K-Means model, cluster-label mapping, and Random Forest classifier) at startup in approximately 1.2 seconds on an Intel Core i7 laptop. Per-query inference consists of three sequential operations: metric extraction (language-aware parsing, approximately 180–220 ms depending on file length), preprocessing (StandardScaler transform + PCA projection, < 5 ms), and classification (Random Forest predict_proba over 100 trees, < 10 ms). Total end-to-end inference latency is approximately 195–235 ms—well within the 400 ms threshold considered imperceptible for interactive web applications. The interface presents the quality label in a colour-coded badge (green / amber / red), the top three contributing metrics ranked by their importance-weighted deviation from the Good-tier mean, and a probability bar showing the model’s confidence across all three classes.

Table 2 — System Evaluation Summary

Evaluation Dimension	Result / Outcome
GitHub corpus size	4,000 files (2,000 Python + 2,000 JavaScript) with MD5 deduplication
Feature extraction stability	20 metrics extracted per file; 10-second per-file timeout eliminates blocking
Optimal k (K-Means)	k = 3 confirmed by elbow method; inertia drop from k=2→3: 35.9%
Label distribution (Python)	Good: 1,082 (54.1%) Average: 614 (30.7%) Bad: 304 (15.2%)
RF accuracy (Python / JS)	99.2% / 98.9% on 20% stratified hold-out (random_state=42)
Top-3 features by importance	Cyclomatic Complexity (0.218), Nesting Depth (0.162), Comment Ratio (0.114)

Inference latency (Streamlit)	195–235 ms end-to-end on commodity CPU (< 400 ms perceptibility threshold)
PCA variance retained	87.3% of total variance in 10 principal components from 20 metrics

Limitations: (1) Cluster-derived labels reflect data-driven regularities, not expert ground truth; near-perfect accuracy should not be misconstrued as alignment with human reviewers. (2) Static metrics cannot detect semantic correctness, algorithmic efficiency, or architectural appropriateness: a structurally clean but semantically incorrect program may be rated Good. (3) The system is currently restricted to Python and JavaScript; extending to Java, Go, or TypeScript requires language-specific parsing extensions. CodeGuardian AI is appropriately positioned as a complement to, not a replacement for, human code review and test-driven quality assurance.

6. CONCLUSION AND FUTURE WORK

This paper presented CodeGuardian AI, a hybrid unsupervised–supervised machine-learning pipeline for automated, multi-language code quality assessment. The system collects 4,000 real-world GitHub source files, extracts twenty static-analysis metrics across four quality dimensions, discovers three natural quality tiers through K-Means clustering anchored by cyclomatic complexity, and trains five supervised classifiers of which Random Forest achieves the highest accuracy (99.2% Python, 98.9% JavaScript). The trained pipeline is deployed as a Streamlit web application with sub-235 ms end-to-end inference latency on commodity hardware. Feature importance analysis identifies cyclomatic complexity, nesting depth, and comment ratio as the three most predictive metrics, confirming that structural and documentation dimensions jointly determine holistic code quality.

Four future enhancements are planned. First, incorporating a small expert-labelled validation set (target: 200–500 files with at least three independent reviewer ratings) would enable calibration of the cluster-derived labels against human ground truth, quantifying the alignment between data-driven and expert-assessed quality. Second, extending the metric extractor to Java, TypeScript, Go, and Rust via language-specific AST parsing would expand the system’s applicability to a wider range of industrial codebases. Third, augmenting the feature set with abstract-syntax-tree-derived structural patterns—function call depth, class inheritance depth, coupling between objects—would capture architectural quality signals that static line-level metrics cannot encode. Fourth, integrating Transformer-based code embeddings (CodeBERT, GraphCodeBERT) as complementary high-dimensional features alongside the twenty static metrics would enable the classifier to leverage semantic contextual information, potentially reducing the rate of false positives on structurally simple but semantically rich code. Fifth, exposing the system as a CI/CD-callable REST API via FastAPI would enable integration into automated quality gates in GitHub Actions, GitLab CI, and Jenkins pipelines without requiring the Streamlit web interface.

➔ **Reproducibility:** The complete CodeGuardian AI pipeline is implemented in Python 3 using pandas, NumPy, scikit-learn Lizard, and Streamlit— all freely licensed and installable via pip. No GPU, cloud API, or commercial license is required. All random seeds are fixed (random_state=42 throughout), and the MD5 deduplication ensures a deterministic corpus given the same search keywords.

7. REFERENCES

- [1] R.C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall, 2008.
- [2] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, “Analyzing the state of static analysis: A large-scale evaluation in open source software,” in Proc. IEEE Int. Conf. Software Analysis, Evolution and Reengineering (SANER), pp. 470–481, 2016.
- [3] T.J. McCabe, “A complexity measure,” IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, pp. 308–320, 1976.

- [4] T. Yin, “Lizard: A simple code complexity analyser,” GitHub repository, 2012. [Online]. Available: <https://github.com/terryyin/lizard>
- [5] Streamlit Inc., “Streamlit: The fastest way to build and share data apps,” 2019. [Online]. Available: <https://streamlit.io>
- [6] J. MacQueen, “Some methods for classification and analysis of multivariate observations,” in Proc. 5th Berkeley Symposium on Mathematical Statistics and Probability, pp. 281–297, 1967.
- [7] L. Breiman, “Random forests,” Machine Learning, Vol. 45, No. 1, pp. 5–32, 2001.
- [8] F. Pedregosa et al., “Scikit-learn: Machine learning in Python,” Journal of Machine Learning Research, Vol. 12, pp. 2825–2830, 2011.